Wolfram for data processing and visualization

Stijn Schildermans
ES&S-DTAI
KU Leuven
Diepenbeek, Belgium
stijn.schildermans@kuleuven.be

Kris Aerts
ES&S-DTAI
KU Leuven
Diepenbeek, Belgium
kris.aerts@kuleuven.be

Abstract

Wolfram is a modern multi-paradigm, mainly functional programming language with a strong focus on user-friendliness and intuitive use. Therefore it is highly declarative in nature and focuses on a transparent and uniform syntax. Furthermore, Wolfram possesses several unique features that greatly add to the scope of the language and simplify tasks such as software deployment.

This paper focuses on the use of Wolfram as a functional programming language for data processing and visualization. The main goal is to provide insight in the benefits and weaknesses of this language compared to other (functional) technologies for these specific tasks. This is done via a case study in which Wolfram code for a specific task is compared to analogue solutions in different languages.

During the case study it quickly became apparent that the Wolfram syntax is unique and intuitive to use. Because of its very declarative nature complex tasks often take only a few lines of code, and are easy to perform. On the other hand, issues with interpretation of code started to appear when Wolfram was used for more complex tasks. The performance of Wolfram also appeared to be sub-par compared to other popular programming languages. Therefore, this case study determined that Wolfram is a very suitable language for quickly creating small applications for data processing and visualization, but for complex and computationally expensive applications other languages might be more suitable.

Keywords Wolfram, Case study, Performance, Data processing, Data visualization

ACM Reference format:
Stijn Schildermans and Kris Aerts. 2017. Wolfram for data processing and visualization. In *Proceedings of ACM SIGPLAN Conference on Programming Languages, Bristol, UK, August 30–September 1, 2017 (IFL'17),* 8 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Wolfram [6] is a modern programming language that aims to combine features from several programming paradigms to create an intuitive programming language that can be used for quickly and easily performing a multitude of tasks ranging from data processing to creating interactive web forms. Wolfram also possesses many interesting features that can not be found in any other programming language.

One application domain in which an intuitive and functional programming language might prosper is that of data processing and visualization. This is the case because the functional programming mindset -viewing the program as a transformation from a certain input to a certain output through the application of first-order functions- converges perfectly with the needs of this application domain. Therefore this paper aims to explore the specific benefits and weaknesses of using Wolfram for this task.

As an extension to this core objective this paper also explores some of the features of Wolfram that go beyond those of most other programming languages and are useful for the studied application domain. After all, Wolfram is mostly known from the search engine Wolfram|Alpha, which is directly integrated in the language. Wolfram also provides an integrated cloud platform, which promises to simplify the deployment process greatly. It is evident that these features should be taken into account when forming an opinion about the suitability of Wolfram for the task of data processing and visualization.

Thus, the core goal of this paper is to form a judgment on the value of Wolfram as a platform for performing tasks such as data processing and visualization, considering the syntax and properties of the language itself, as well as the additional features the language provides that can not be found in alternative programming languages.

This paper is based on the main author's Master's thesis [3]. The case study discussed in this paper is described in more detail in said thesis.

2 Background: the Wolfram language

The Wolfram programming language originates from Mathematica [8], which is a commonly used platform for scientific computation and modelling. Although the language is syntactically radically different from the commonly used programming languages today, it is easy to use and intuitive. This section gives a brief overview of the Wolfram syntax and design principles, as well as the most important unique features it possesses that form an added value for the task of data processing and visualization.

IFL'17, August 30–September 1, 2017, Bristol, UK
2017. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

The official Wolfram language reference documentation [7] is very complete and concise. This documentation forms the main source for this section. Interested readers are encouraged to explore the language further via this medium.

2.1 Syntax

In Wolfram, everything from a mathematical operation to a complex geometrical figure is displayed as a uniform symbolic expression. These expressions are formed by a head which serves as a name, and a set of arguments. Expressions are used for declaring constants as well as variables and functions.

Wolfram supports two kinds of assignments: immediate assignment and delayed assignment. Immediate assignments are calculated immediately and denoted using '=' as in 'a = 2', while delayed assignments are recalculated every time the assigned expression is evaluated. Delayed assignments are denoted with ':=' as in 'b := 5'.

Functions in Wolfram are just expressions that were assigned using the delayed assignment method and take an argument list. There is no further distinction between functions and other data in Wolfram. This is a good example of how Wolfram considers functions as first class citizens. The fact that constants and functions can have exactly the same notation is mathematically pure and a strong functional feature of the language. The code sample below illustrates the principles described above.

```
a = 2;
b := 5
c[x__] := 5
d[x__] := Max[x]
```

The expressions a and b are resp. an immediate and delayed assigned constant. The expressions c and d illustrate that constants and functions can have exactly the same notation, which is mathematically pure and is a strong functional feature of the language.

Wolfram is partly because of its functional nature a highly declarative language. Code is interpreted flexibly by the run time environment which partially compensates for mistakes by the programmer. Furthermore, the language is dynamically typed to further simplify the programming process.

Wolfram also incorporates many well known features from the functional programming paradigm. One example is the possibility to map functions on lists. Wolfram provides several ways of doing this, and even gives the programmer the possibility to use very compact syntax for this common task. The code sample below illustrates this.

```
Map[f, {a, b, c}]
f /@ {a, b, c}
```

The lines of code illustrated above have the same effect. Both map the function f on the list {a,b,c}. Wolfram has many features like this for applying functions directly to complex data structures. Interested readers are referred to [4].

2.2 Features

Besides the elegant and compact syntax the Wolfram language has several unique features that make tasks like data processing and visualization among others a lot easier than in other languages. This section summarizes the most important ones for the studied application domain.

2.2.1 Wolfram|Alpha

The Wolfram language contains several built-in functions that allow the developer to directly interact with the Wolfram|Alpha search engine programmatically. This is a very powerful tool that can be used to collect relevant side information about all kinds of data and real-world entities. This feature can be very useful for data processing. The code below illustrates this.

```
Bristol["Population"]
```

Wolfram|Alpha is used to interpret 'Bristol' in the code sample above. This results in a real-world entity of the type 'city'. This can be interpreted as an object with a set of properties that can be queried. In this example, the property 'Population' is extracted from the real-world entity. The result of this code is '60 147 people'.

Note that anything that has to be interpreted by Wolfram|Alpha has to be entered into a special search field that can be invoked anywhere in the code by pressing 'ctrl + enter'. This also implies that Wolfram code is best always written using a dedicated IDE, i.e. a version of Mathematica[1] or the online Wolfram Development Platform[2].

2.2.2 Data visualization functions

Another interesting feature of the Wolfram language is that it contains several built-in functions that make generating complex graphics from a data set a matter of a few lines of code. Functions like *ListPlot* generate graphs from a list of input data very easily. The code below generates a simple list plot from some arbitrary input data.

[1] https://www.wolfram.com/mathematica/
[2] https://www.wolfram.com/development-platform/

```
data = {{1,1},{2,5},{3,1}};

ListLinePlot[data]
```

Figure 1 shows the result of this code.

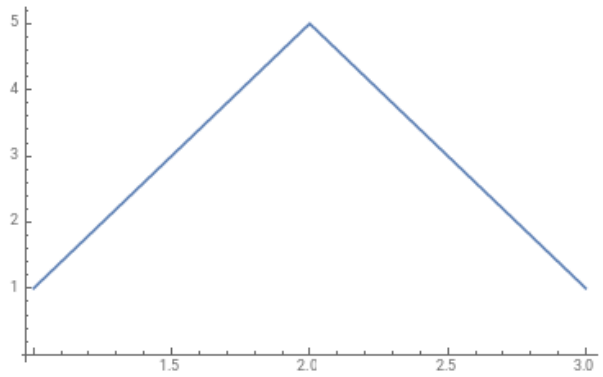

Figure 1. Simple list plot in Wolfram.

The result of this minimal code is a simple graph that already provides all essential features to interpret the data. Through the use of so called 'options' these graphics can be customized further in many ways. It is possible to add plot labels, a legend, a custom color scheme, etc. The customization options are numerous and thorough. This again shows the highly declarative nature of Wolfram. Thanks to this approach the code is kept as compact as possible while maintaining possibilities for customization.

2.2.3 Wolfram Cloud

A final feature of the Wolfram Language that is interesting for any type of application, is the possibility it provides to instantly deploy any Wolfram code to the integrated Wolfram Cloud through the use of the built-in function *CloudDeploy*. Wolfram also has dedicated functions for creating RESTful web APIs, and even embedding cloud-deployed Wolfram code directly in other web pages. For example, the code below generates a simple *Listlineplot* like in section 2.2.2, but this time this plot is deployed to the cloud in the form of an API that takes a sequene of integers as input, and returns a graph corresponding to the input as a png image.

```
CloudDeploy[APIFunction[
"d"->DelimitedSequence["Integer"],
ExportForm[ListLinePlot[#],"PNG"]&]]
```

The code above shows just how compact and declarative the Wolfram language is for quickly creating APIs for data visualization. The returned image can be directly embedded in any other web page or program. Wolfram even provides functions that generate the code required for this.

Note that the input to the *ListLinePlot* function is a list of numbers in this case, rather than a nested list of x,y-coordinates like in section 2.2.2. Wolfram automatically interprets a plain list in this case as a list of x-y coordinates, with the x-coordinates being an ascending sequence of natural numbers starting at 1. Thus, the resulting list plot will in this case be identical to the results from section 2.2.2, provided the parameter *d* gets the value '1,5,1'. This is a great example of the dynamic way in which Wolfram interprets code, which often makes the task of quickly creating simple services for data processing and visualization very easy compared to other programming languages.

3 Method

As stated above the main goal of this paper is to determine whether Wolfram is a suitable language for data processing and visualization, and why Wolfram should or should not be chosen for this task above other languages. The method used to determine this is a case study. Since this paper is based on the Master's thesis of the main author, said case study directly originates from [3].

In [3] several programming languages are compared by writing a recursive algorithm that generates Pascal's triangle. This is a mathematical structure in which numbers are arranged in a triangular shape, and each element is the sum of the two elements directly above it, with all the 'border elements' having the value '1' [1]. This is demonstrated in figure 2.

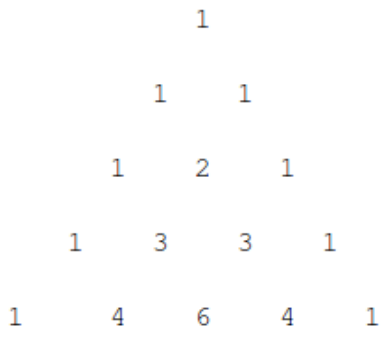

Figure 2. Pascal's triangle [1].

The studied languages are Java, C#, F# and Wolfram. Since this paper only covers Wolfram, only the results for Wolfram will be discussed in detail, and the other languages will be used as a reference. All implementations are then compared based on several predetermined parameters, being:

- Readability,
- Code length,
- Ease of use,
- Performance.

The performance of the studied languages in [3] was compared from within Wolfram using the J/Link [9] and .NET/Link [5] libraries. These libraries start an independent Java resp. .NET run time environment on the local machine and import the results in Wolfram for further processing. In this case this allowed for a uniform timing of the performance of the various tested languages. From within Wolfram the Pascal's triangle algorithm is called for different sizes of the triangle, and the time needed to compute the entire triangle is measured programmatically for each implementation. This is repeated for 10 iterations. The average time needed for generating the triangles is calculated and visualized using Wolfram.

Both the code for generating Pascal's triangle -requiring a lot of computation- and for processing and comparing the timings, are use cases for determining the suitability of Wolfram for data processing and visualization tasks. Appendix A provides the full Wolfram script that processes the data from the performance tests. The function takes a list of language names as input and custom decoder functions were written to link these names to the actual data to be generated and the function to be used for curve fitting in the visualization step. The result of this is a plot such as figure 3.

4 Results

This section discusses the results of the comparative case study described above. The results are divided into several major sections that are discussed in detail. Finally, a general overview is given and a direct comparison is made with the languages Java, C# and F#.

4.1 Implementation

This section discusses implementation-oriented criteria concerning the suitability of Wolfram for the tasks of data processing and visualization. Several advantages of Wolfram are illustrated, as well as some drawbacks.

4.1.1 Intuitive and concise code

As stated in section 3, this case study is based upon the aforementioned recursive Pascal's triangle algorithm. This can be seen as an algorithm that generates/processes data and can be identically implemented in many languages, thus making a transparent comparison possible. The said algorithm is very easy to implement in Wolfram. The code below shows the implementation used for this research.

```
ptr[s_Integer] /; s == 1 := {{1}}
ptr[s_Integer] :=
  Table[ptrv[i, j], {i, 1, s},
    {j, 1, i}]
ptrv[r_Integer, c_Integer]
  /; c == 1 || c == r := 1
ptrv[r_Integer, c_Integer] :=
  ptrv[r - 1, c - 1] + ptrv[r - 1, c]
```

The code above is a good illustration of the compactness of the Wolfram language. Like many functional languages it uses pattern matching for determining control flow as a compact and elegant alternative for if-statements from the imperative programming world. This results in very readable and concise code. Furthermore the declarative and flexible nature of the language is clear from the use of the Table function for generating a nested list in which the length of the inner lists is dependent on the index of the outer list; a task that is relatively complicated in imperative languages and usually requires two nested for loops. The code below shows a reference implementation in Java with the same functionality.

```
public static int[][]
pascalTriangle(int size) {
    int[][] points = new int[size][];
    for (int i = 0; i < size; i++) {
        int[] row = new int[i + 1];
        for (int j = 0; j <= i; j++)
            row[j] =
            getValueAtPoint(i, j);
        points[i] = row;
    }
    return points;
}
public static int
getValueAtPoint(int row, int col) {
    if (col == 0 || col == row)
        return 1;
    else return
    getValueAtPoint(row - 1, col - 1)
    + getValueAtPoint(row - 1, col);
}
```

It is clear that the Java version is much longer, and contains much more clutter and syntactical elements. Also the Java version is much less readable, and the many variables that have to be explicitly assigned and read make it more challenging to get the algorithm error-free.

The compact and intuitive syntax might set Wolfram apart from imperative languages, but many other functional languages like Haskell and F# share these properties. However, generally during this research Wolfram appeared to possess the most compact and intuitive syntax of all tested languages, and this without sacrificing readability. Below is the F#-equivalent of the recursive Pascal's triangle algorithm for comparison.

```
let rec pascalTriangleValue row col =
match col with
    | 1 -> 1
    | col when col = row -> 1
    | _ -> pascalTriangleValue
           (row-1) (col-1)
           + pascalTriangleValue
           (row-1) col
let pascalTriangle size = [1..size]
    |> List.map (fun r->[1..r]
    |> List.map
    (fun c->pascalTriangleValue r c))
```

The F# code is a lot more compact and readable than the Java version, but is still longer than the Wolfram code. Also, it is clear that this code adheres much more to classic functional principles, while Wolfram takes its own, unique approach. In conclusion to this section, it can be said that Wolfram has a unique, very compact and very declarative syntax that is greatly beneficial for tasks like data processing.

4.1.2 Unexpected behavior

The many design choices of the language that are oriented to ease of use have an effect on the overall clarity of what the code is exactly doing, which can sometimes lead to unexpected behavior. A good example is the dynamic typing system of Wolfram. In combination with the flexible interpretation of code this means that Wolfram code almost never crashes, but this also means that functions even work on data that is not suitable for that function, and just return an unexpected result. This can be easily demonstrated by a simple code example:

```
AllTrue[{{1}}, ListQ]
AllTrue[{1}, ListQ]
AllTrue[1, ListQ]
```

The function AllTrue expects a list and a function that returns a boolean as input, and returns a boolean depicting whether or not all the elements in the list comply with the boolean test function. The function ListQ assesses whether or not its input is a list.

In the first expression in the code sample above, the input is a list with one element, being itself a list with one element. The expression thus evaluates to 'True' as expected.

In the second expression however, the list of input arguments contains only the number one, which is not a list, so the expression evaluates to 'False'.

In the third expression, the first input element is not even a list, so most other programming languages would throw an exception at this point. Wolfram does not however, and reinterprets the input '1' so that the expression does evaluate successfully. Strangely though, this expression evaluates to 'True', while intuitively 'False' should be the correct answer. This demonstrates that the great flexibility of Wolfram might indeed lead to unexpected results, and this can happen quite easily. For complex programs this might be a serious concern, as behavior like this might induce errors that are difficult to find. Luckily, Wolfram contains several functions that can give the programmer stricter control over the evaluation of code[3].

4.2 Powerful features

What really sets Wolfram apart from the other functional languages for the task of data processing and visualization are the built-in features described previously in this paper. The instant access to the wealth of information that Wolfram|Alpha provides and the flexibility of the Wolfram language to interpret this information can be of great value when processing certain kinds of data. One example that proved particularly useful during this research is the flexible way Wolfram can interpret units of time. Dates and times can be directly subtracted from each other and instantly used in graphs and other applications. All formatting is abstracted from the programmer, and Wolfram|Alpha is used for many conversions behind the scenes. The code below illustrates this feature.

```
t = Now - 1 wk - 5 min
```

[3] http://reference.wolfram.com/language/guide/EvaluationControl.html

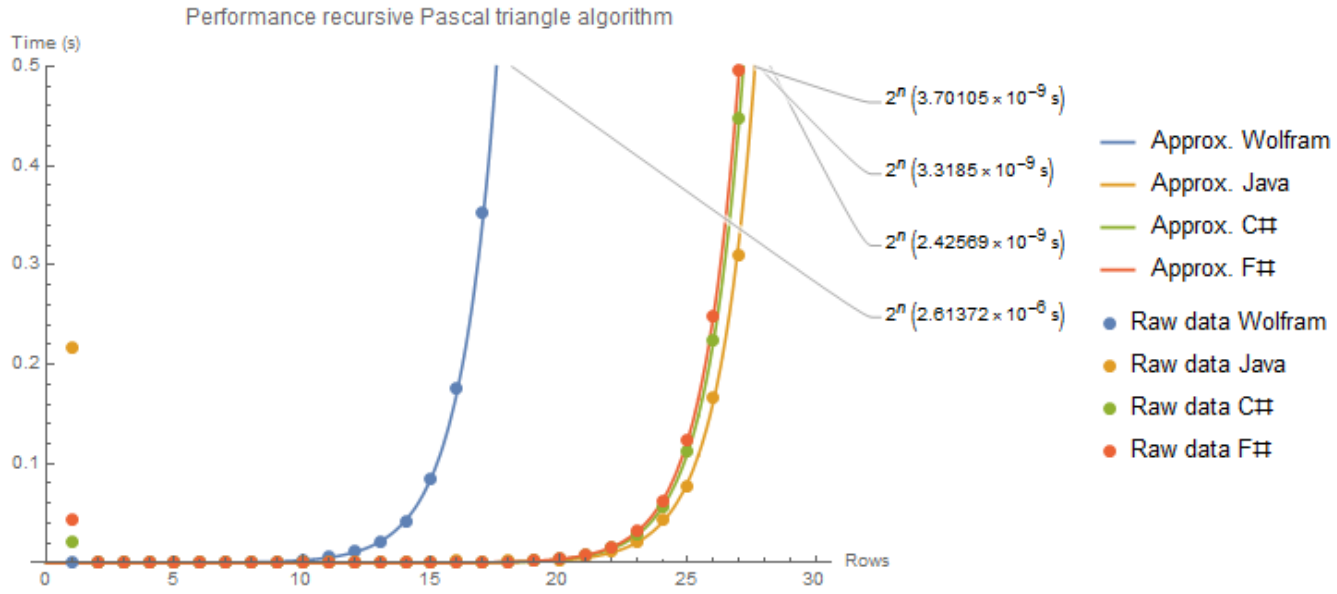

Figure 3. Complex plot in Wolfram.

The result of this code is a DateTime expression that represents the point in time 1 week and 5 minutes before the variable t was assigned. This DateTime expression can be used seamlessly in other code. Wolfram|Alpha interprets the terms 'wk' and 'min' and converts them to the appropriate form. This makes working with dates and times much more intuitive in Wolfram than in other languages.

The features described in section 2.2 have also proven to be very useful for this research. The intuitive data visualization functions made visualizing test results fast and easy to implement. Deploying Wolfram code was in this case study also remarkably easier than deploying programs written in any other programming language, thanks to the integrated Wolfram Cloud. This is a major advantage for creating quick services for processing and visualizing data, that can be easily integrated into other applications as demonstrated in section 2.2.3.

4.3 Elegant and flexible data visualization

This section discusses the experiences during this case study for visualizing the results of the performance tests discussed in section 3. The code used to visualize the results of the performance tests in this case study is shown in appendix A. In this example input data is first processed, and then displayed in a plot, together with an approximation of the data generated by curve-fitting. Labels, a legend, etc. were all added by means of options, as discussed in section 2.2.2. The result of this code is shown in figure 3.

As can be seen in figure 3, these options are a powerful tool with many possibilities. Each data visualization function has many options that allow the programmer to do virtually anything he would reasonably want to do in a very compact and intuitive way.

4.4 Performance

The performance of Wolfram was tested and compared with that of Java, C# and F# using the method described in section 3. Wolfram turned out to be far slower than any of its competitors. Table 1 shows the relative performance of Wolfram compared to the other languages tested in [3]. The resulting numbers are a factor that indicates the time needed by Wolfram to perform this task compared to the time needed by the other languages.

Table 1. Relative performance of Wolfram compared to the other languages tested in [3].

Language	Wolfram
Java	1070
C#	800
F#	700

These results indicate that Wolfram is indeed much slower than any of the other tested languages in [3]. This has serious repercussions for the application domain of data processing and to some extent also visualization. Wolfram turns out to be 1000 times slower than Java, which is of course unacceptable for any application where hardware resources are limited, or where performance is of any kind of importance.

4.5 General

Table 2 shows a comparison between Wolfram and all the other tested programming languages in [3] based on several parameters discussed in this case study. The table depicts a qualitative judgment, relative to the other languages.

Table 2. Comparison between Wolfram and several other programming languages [3].

Language	Readability	Length	Ease of use	Performance
Java	OK	20	Good	Excellent
C#	OK	25	Good	Good
F#	Good	10	Bad	Good
Wolfram	Good	5	Excellent	Poor

Compared to the other tested languages Wolfram scores very well in the implementation-oriented categories. The only major weakness of Wolfram is performance. The very high-level approach of Wolfram and the great ease of use that is induced by it clearly have a cost when it comes to performance. Also the fact that Wolfram is an interpreted rather than a compiled language can contribute significantly to its poor performance [2]. This might be a serious concern for many data processing tasks, although the many advantages of the language might compensate for this fundamental problem in many cases.

It is also important to note that the compact, typeless syntax in combination with the flexible interpretation of Wolfram code might lead to unexpected results as described in section 4.1.2. This is especially a drawback for large and/or complicated programs and can lead to bugs that are difficult to detect.

5 Conclusion

Wolfram is a modern programming language that takes a unique approach to software development through its very compact and uniform syntax, which encompasses many functional features as well as aspects from other programming paradigms. Thanks to the declarative and functional nature and several unique features such as very easy cloud deployment and integration with Wolfram|Alpha, Wolfram is an excellent language for applications such as data processing and visualization from an implementation standpoint, although for large projects the dynamic typing system and flexible interpretation of code might be counter productive at times.

The largest drawback of Wolfram for the studied application domain however is its very poor performance. This case study showed relative performance numbers that are not acceptable for computationally expensive tasks compared to other popular programming languages; both functional and object-oriented.

Wolfram is thus a powerful tool for quickly creating and deploying applications or services for data processing and visualization, as long as the task at hand is not too large in scope or too computationally expensive. However, when performance is an issue, other programming languages might be more suitable.

Acknowledgments

Special thanks Tom Schrijvers who was closely involved in the constitution of this paper and provided comprehensive and vital feedback.

References

[1] 2017. Pascal's Triangle. (2017). http://mathforum.org/workshops/usi/pascal/pascal_intro.html

[2] John S. Riley. [n. d.]. Interpreted vs. Compiled Languages. ([n. d.]). http://dsbscience.com/freepubs/start_programming/node6.html

[3] Stijn Schildermans. 2017. *Exploratie van functionele en declaratieve ontwikkelmethodes voor cloud programming.* Master's thesis. UHasselt.

[4] Wolfram. [n. d.]. Applying Functions. ([n. d.]). https://www.wolfram.com/language/fast-introduction-for-programmers/en/applying-functions/

[5] Wolfram. 2017. .NET/Link User Guide. (2017). http://reference.wolfram.com/language/NETLink/tutorial/Overview.html

[6] Wolfram. 2017. Wolfram: Computation Meets Knowledge. (2017). http://www.wolfram.com/

[7] Wolfram. 2017. Wolfram Language and System Documentation Center. (2017). http://reference.wolfram.com/language/

[8] Wolfram. 2017. Wolfram Mathematica. (2017). https://www.wolfram.com/mathematica/

[9] Wolfram. 2017. Writing Java Programs That Use the Wolfram Language. (2017). http://reference.wolfram.com/language/JLink/tutorial/WritingJavaProgramsThatUseTheWolframLanguage.html#15141

A Wolfram code for processing results of performance tests

```
testPerformance
[langs_List, iterations_Integer] :=
testPerformance[langs,iterations,0.5]
testPerformance[langs_List,
iterations_Integer, yRange_]:=Catch[
testData = Table[ReleaseHold[#],
{i, iterations}] &
/@ ((data[#]) &) /@ langs;
avg = ({First[First[#]],
Mean([#[[2]] &) /@ #]} &)
/@ Map[Flatten,
Table[(Cases[#, {i, _}] &)
/@ #, {i, Length[#[[1]]]}], {2}]&
/@ testData;
listPlot = ListPlot[avg,
PlotStyle -> PointSize[Large],
PlotRange -> {0, yRange},
PlotLegends -> ("Raw data " <> # &)
/@ langs,
TargetUnits -> "Seconds"];
tupels = Table[{avg[[i]],
fittingFunction[langs[[i]]]},
{i, Length[langs]}];
fit = Fit[(#[[1]])[[2;;]],
#[[2]], n ] & /@ tupels;
plot = Plot[Evaluate[fit],
{n, 0, Max[Length[#[[1]]] &
/@ testData]},
PlotLabel ->
"Performance Recursive Pascal Triangle",
PlotRangeClipping -> False,
PlotRange -> {0, yRange},
AxesLabel -> {"Rows", "Time (s)"},
PlotLegends -> ("Approx. " <> # &)
/@ langs,
PlotLabels -> fit];
Show[plot, listPlot,
ImageSize -> Large],
InvalidInputException];
```